intel®

APPLICATION
NOTE

AP-239

November 1986

Customer Applications of the EMV-88 Emulations Vehicle

BILL ALLEN
DSO PRODUCT MARKETING
FRED MOSEDALE
DSO TECHNICAL PUBLICATIONS

Order Number: 280105-001

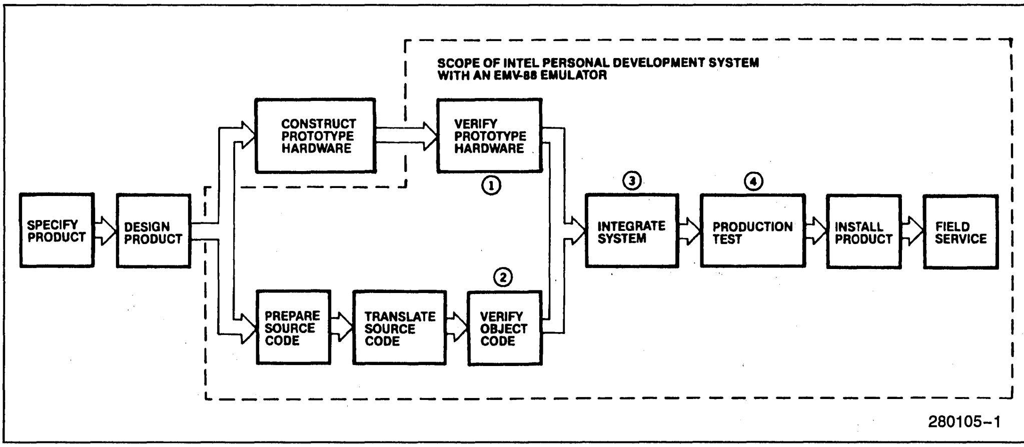

Figure 1. Typical Microcomputer Process

INTRODUCTION

Early customers' experiences with the EMV-88 emulation vehicle have shown the power, versatility, and benefits of this emulator. The EMV-88 emulator plugs into an Intel Personal Development system (iPDS™) and aids in the development and debugging of user-designed 8088 systems.

To aid new and potential users of the EMV-88, this application note summarizes applications and debugging procedures of several early users of the EMV-88 emulator.

THE MICROCOMPUTER DEVELOPMENT PROCESS

Designing a product that contains a microcomputer requires close coordination of two separate but highly dependent design efforts: hardware development and software development.

Hardware development involves planning the microprocessor chip's interaction with associated logic, memory, peripheral circuits, and specialized circuits.

Software development involves programming the microcomputer system to perform the required tasks. The resulting program eventually resides in the product's memory.

These two development efforts can be accomplished independently, but it is more efficient to work on them together. Successful designs make maximum use of the hand-in-hand nature of hardware and software. In addition, real-world designing is an iterative process. Each step in the design process may involve the debugging and re-design of previous steps. A change in hardware may involve a corresponding change in software and vice versa.

Figure 1 illustrates a typical microcomputer development process. The iPDS system and the EMV-88 emulator are two major design aids that Intel offers to hardware and software designers. Note the areas of the design process where a microcomputer development system and an emulator aid in the design of a product. (Numbers 1 through 4 shown in Figure 1 are used later in this application note.)

Features of the iPDS™ System and the EMV-88 Emulator

The iPDS system and EMV-88 emulator offer the following resources.

- A stand-alone computer with dual processors (optional), memory, mass storage, and a disk-based operating system.
- Development system software such as assemblers, high-level language compilers, and EMV-88 debugging software.
- An interface (using the EMV-88 cable) to the prototype hardware. This allows you to check each piece of your prototype hardware as it is developed.
- The EMV-88 mapping capability. This allows you to borrow the memory form the EMV-88 until prototype hardware is available.
- The EMV-88 break and trace ability. This allows you to specify the conditions under which emulation stops or tracing occurs.

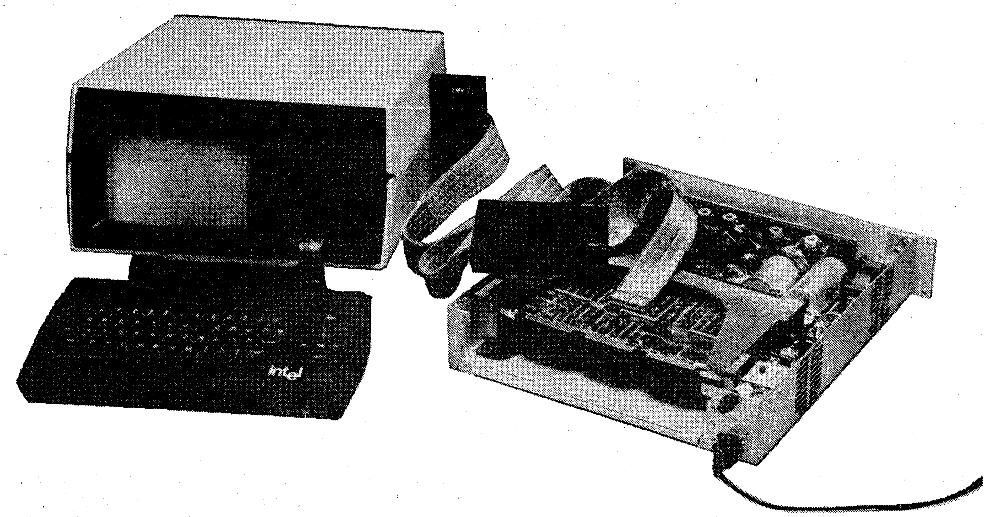

Figure 2. The iPDS™ System and the EMV-88 Emulator Connected to a User Prototype

- Availability of other plug-in modules for PROM programming and for emulation of other processors (8051, 8044, and 8085).

Figure 2 shows the EMV-88 emulator inserted in the side of the iPDS system. The EMV-88 emulator is shown connected to a user prototype board.

THE USERS

To obtain information for this application note, three early EMV-88 users were visited. Users were asked a variety of questions: How did the EMV-88 system save them time? What debugging procedures were especially useful? What EMV-88 features proved important in developing the users' designs? In retrospect, what features or techniques might have been used earlier to speed development and debugging even more?

The customers visited were the following:

— An established company originally specializing in water and waste water control devices. Now it is expanding to provide automatic control and monitoring devices. This company recently began designing microcontrollers and microprocessors into its products.

— An established communications and antenna company that was designing an 8088-based system to control the antenna for a satellite communications system.

— A new company that was designing a computerized system for monitoring automatic manufacturing process control machines.

All the customers had iPDS systems with EMV-88 plug-in emulators. (The customers also had other iPDS plug-in options, including PROM programming modules.)

USER APPLICATIONS AND DEBUGGING PROCEDURES

New and potential EMV-88 users will be interested in two kinds of information supplied by the early EMV-88 users:

(1) the different functions the EMV-88 emulators can be used to perform in early phases of a product's life cycle, and

(2) some specific EMV-88 debugging techniques that proved useful to the users. The following two main sections focus on these topics.

EMV-88 Emulator Functions in Early Phases of a Product's Life Cycle

As a product is developed, debugged, and released to customers, the EMV-88 emulator can accomplish a variety of tasks during early phases of the product's life cycle. In particular, the iPDS system with the EMV-88

emulator proved to be especially productive in completing the following tasks. (The task numbers are also shown in circles in Figure 1.)

Task 1 Verifying hardware

Task 2 Verifying software

Task 3 Integrating prototype hardware and software

Task 4 Production testing

The EMV-88 system helped early users accomplish these tasks as described below:

- **Exercised hardware and software in real time** (tasks 1, 2, and 3): Without an emulator, users would only have been able to check their prototypes by loading their programs into EPROMs and running the programs. Then, when bugs were detected and corrected, the revised programs would have to be loaded into the EPROMs. This cycle would have to be repeated each time a bug was found. However, with the EMV-88 emulator, users did not need to load programs into EPROMs. The program resided in emulator memory and could easily be modified and retested. Thus, the emulator saved users time and provided flexibility in modifying programs.
- **For a prototype system with large programs, the EMV-88 emulator was used to supplement main development systems** (task 1): One user had very large programs under development. The user's large development systems were tied up with software development and could not be used with their emulators to test hardware. To speed up debugging, the iPDS system with its EMV-88 emulator was pressed into service. Short EMV-88 macros were written to exercise particular portions of the hardware. Thus, hardware development and testing could continue despite the unavailability of emulators on the large development systems.
- **Patched around missing sections of code to allow emulation** (tasks 2 and 3): Because the EMV-88 emulator allows users to patch code, whenever a section of code is incomplete or contains a bug, users can patch around it. One user was able to begin debugging the prototype before software development was completed. At the end of sections of completed code, EMV-88 commands were used to patch in a command to jump from the last line of code in one section to the first line of code in the next available section. Some of the activities of the still-to-be-completed code were also simulated with EMV-88 commands in the patch.
- **Resolved disputes about whether bugs were in hardware or software** (task 3): Because the iPDS system and its EMV-88 emulator can control and examine both user hardware and user software, it is relatively easy to determine whether a bug originates in software or hardware. For example, users took advantage of the EMV-88 emulator's single-stepping capability to determine at which line of code undesired values were generated. Then, carefully controlled emulation combined with the use of a logic analyzer allowed users to pinpoint the source of the problem. As a result, fingerpointing by software and hardware team members quickly came to an end.
- **Provided remote site testing of prototype hardware and software** (task 3): One user could not fully test the prototype hardware and software because the environment in which the prototype was intended to run could not be duplicated in the development area. Because the iPDS system and the EMV-88 are portable, they were easily moved to a site remote from the development area. Then, a full debugging session in the prototype's intended environment took place.
- **Tested early manufactured systems** (task 4): When the earliest boards have been manufactured, there must be a way to test them. If a complete board-testing system is not yet in place, the EMV-88 can act as a hardware tester. After test programs are written for the EMV-88, and the emulator is connected to a new board, users can quickly determine whether flaws exist in the manufacturing process. If tests are skillfully written, hardware areas that are failing can be pinpointed.
- **Used to troubleshoot early customer systems** (task 4): Despite careful quality control, not all bugs in the design and manufacturing processes may be detected. Early customers may report problems they are having with the product. If swapping hardware and/or software corrects the problem, the defective hardware and/or software can be returned for troubleshooting. The EMV-88 emulator can track down the problem. It is important to determine the source of the problem so that, if need be, design changes or manufacturing changes can quickly be initiated.

As is evident from the preceding list, early users found a variety of tasks for the EMV-88 emulator to perform during product development and manufacturing. One user noted that the iPDS system is an excellent development system for both young and mature companies. Its low price, versatility, and portability make the iPDS system with the EMV-88 emulator an investment that returns its cost many times.

Early EMV-88 Users' Debugging Techniques

Six debugging problems early users encountered have been selected to illustrate a variety of EMV-88 emulator's capabilities.

```
*DEFINE :move          ;Macro name is :move (* is the EMV-88 prompt)
.*BASE=Y               ;Sets display radix to binary
.*SUFFIX=Y             ;Sets input radix to binary
.*BYTE 0 TO 0111=0     ;Initializes first 8 bytes to 0
.*BYTE 0 TO 0111       ;Displays first 8 bytes
.*DEFINE .n=0          ;Sets memory location variable to 0
.*DEFINE .k=1          ;Sets memory content variable to 1
.*REPEAT               ;Begins first repeat loop
.*UNTIL .k=100000000   ;Halts first loop when .k=100000000
.*REPEAT               ;Begins second loop
.*UNTIL .n=1000        ;Halts second loop when .n reaches 8 (decimal)
.*BYTE .n=.k           ;Sets memory location .n=value .k
.*.n=.n+1              ;Increments .n
.*END                  ;Ends second loop
.*BYTE 0 to 0111       ;Displays values in first 8 memory locations
.*.k=.k*10             ;Multiplies .k by 2 (decimal)
.*END                  ;Ends first loop
.*EM                   ;Ends macro
*
```

280105-3

Figure 3. Sample Macro for Testing Memory

PROBLEM 1: INITIAL CHECK OF PROTOTYPE HARDWARE (Task 1: Verify Hardware)

One early user made an initial check of prototype hardware with the EMV-88 emulator. Once the user's RAM, USART, and registers were in place in the hardware prototype, an initial hardware check was scheduled. Were the components installed and connected properly? (To ease hardware-software integration, efforts should first be made to isolate hardware defects independently of the prototype software.)

PROBLEM 1 SOLUTION

The EMV-88 user performed the following steps to check out prototype hardware.

1. Identified the addresses of all hardware elements to be tested.
2. Devised EMV-88 hardware test macros: Macros were created that wrote patterns of 1's and 0's to the memory devices and registers. The macros also were designed to read and display memory and register contents. (See Figures 3 and 4 for a sample macro that writes patterns of 1's and 0's to a small portion of memory. Also, see the appendix for information on EMV-88 commands.)
3. Executed the macros and observed the results on the IPDS system display screen.
4. Identified defective hardware areas: When an output value was different from an input value, the user executed memory interrogation commands (e.g., BYTE, WORD, DUMP) to confirm the location of defective hardware.

PROBLEM 2: WRONG INSTRUCTION EXECUTION SEQUENCE (Task 2: Verify Software)

When this user's prototype program was emulated, a portion of the program ran properly, but then it performed strangely—it "ran in the weeds." How can the EMV-88 emulator locate the area of program code where the execution sequence first begins to go wrong?

```
*:move                    ;Using macro name causes macro to be executed
BYT 00000H=00000000Y 00000000Y 00000000Y 00000000Y
BYT 00004H=00000000Y 00000000Y 00000000Y 00000000Y
BYT 00000H=00000001Y 00000001Y 00000001Y 00000001Y
BYT 00004H=00000001Y 00000001Y 00000001Y 00000001Y
BYT 00000H=00000010Y 00000010Y 00000010Y 00000010Y
BYT 00004H=00000010Y 00000010Y 00000010Y 00000010Y
BYT 00000H=00000100Y 00000100Y 00000100Y 00000100Y
BYT 00004H=00000100Y 00000100Y 00000100Y 00000100Y
BYT 00000H=00001000Y 00001000Y 00001000Y 00001000Y
BYT 00004H=00001000Y 00001000Y 00001000Y 00001000Y
BYT 00000H=00010000Y 00010000Y 00010000Y 00010000Y
BYT 00004H=00010000Y 00010000Y 00010000Y 00010000Y
BYT 00000H=00100000Y 00100000Y 00100000Y 00100000Y
BYT 00004H=00100000Y 00100000Y 00100000Y 00100000Y
BYT 00000H=01000000Y 01000000Y 01000000Y 01000000Y
BYT 00004H=01000000Y 01000000Y 01000000Y 01000000Y
BYT 00000H=10000000Y 10000000Y 10000000Y 10000000Y
BYT 00004H=10000000Y 10000000Y 10000000Y 10000000Y
*
```

280105-4

Figure 4. Sample Display Resulting from Figure 3 Macro

PROBLEM 2 SOLUTION

The user performed the following steps to locate the area of code where code begins defective operation.

1. Emulated the program: The user executed the GO command with the FROM option; the program starting address was entered after FROM. (See the appendix for information on EMV-88 commands.)
2. Examined the trace buffer: The PREVIOUS command was used to scan through the 1K byte trace buffer. (See Figure 5 for a sample display using the PREVIOUS command. In Figure 5, the first 16 instructions in the 1K byte trace buffer are displayed.) The instructions stored at the very beginning of the buffer were incorrect. This implied that the problem was further back in program execution. The instruction address at the beginning of the trace buffer was noted.
3. Set a breakpoint: To make possible examination of the previous 1K bytes of the program execution sequence, the user set an execution breakpoint at the address identified in step 2.
4. Re-emulated. When emulation occurred using the new breakpoint, emulation halted at the point the previous trace buffer started collecting trace information. Now, the new trace buffer contained the preceding 1K bytes of executed instructions
5. Examined the trace buffer: Scanning through the new trace buffer contents the user came upon the program section where the execution sequence went awry. Study of the program section showed a programming error.
6. Patched code: Using the ORG (originate) and ASM (assemble) commands, the user created a patch. (See Figure 6 for a display of sample EMV-88 patching commands.) First the instruction pointer was moved to the location of the defective line of code using the ORG command. Then, the ASM command inserted a jump command to an unused area of memory. Using ORG and ASM, a patch of correct code was created at the unused memory location; the patch included a jump back to the instruction next after the line of defective code.

```
*PREVIOUS 1024T LENGTH 16 ;Displays first 16 instructions in trace buffer
00408H          MOV DS,AX                                   8ED8      PREV
0040DH          MOV AX,0060H                                B86000    PREV
00410H          MOV ES,AX                                   8EC0      PREV
00412H          NOP                                         90        PREV
..DATA_AND_CODE.XLATE
00413H          MOV         DI,0028H                        BF2800    PREV
00416H          MOV         SI,0000H                        BE0000    PREV
00419H          MOV         BX,0052H                        BB5200    PREV
0041CH          CLD                                         FC        PREV
0041DH          CALL        0440H           ; SHORT         E82000    PREV
..DEMO_PROCS.SPOT1
00440H          LODS        BYTE [SI]                       AC        PREV
00441H          XLAT        BYTE [BX]                       D7        PREV
00442H          STOS        BYTE [DI]                       AA        PREV
00443H          LOOP        0440H           ; SHORT         E2FB      PREV
..DEMO_PROCS.SPOT1
00440H          LODS        BYTE [SI]                       AC        PREV
00441H          XLAT        BYTE [BX]                       D7        PREV
00442H          STOS        BYTE [DI]                       AA        PREV
00443H          LOOP        0440H           ; SHORT         E2FB      PREV
00445H          RET                         ; SHORT         C3        PREV
..DATA_AND_CODE.REVERSE
00420H          MOV         CX,WORD 0050H                   8B0E5000PREV
*
```

280105-5

Figure 5. Sample Display Using the PREVIOUS Command

7. Re-emulated: Emulation stopped at the breakpoint set in step 3.
8. Examined the trace buffer: This time the trace buffer showed that program execution followed the correct sequence. Thus, the patch fixed the problem.

PROBLEM 3: DEBUGGING IN A MULTI-TASKING ENVIRONMENT
(Task 2: Verify Software)

Programs that support multi-tasking can be difficult to debug when interrupts arrive that place the current task on the stack while another task is undertaken. One EMV-88 user performed the following steps to overcome this problem.

PROBLEM 3 SOLUTION

1. Cleared the interrupt enable flag: By entering IFF = 0, the 8088 interrupt enable flag was cleared. See Figure 7 for a display of register settings that shows the resulting IFF setting.
2. Emulated the code of interest: The user used the GO command with the FROM option to set a breakpoint and thus control emulation of the desired section of code. (See the appendix for information on EMV-88 commands.) With the interrupt enable flag cleared, trace information was collected without other tasks interrupting the trace data collection.
3. Re-enabled the interrupt enable flag.

PROBLEM 4: MEMORY NOT BEING ZEROED
(Task 3: Integrate Hardware and Software)

This user first employed the EMV-88 emulator to test a section of code that was supposed to zero memory. The test showed that memory was not being zeroed. What prevented the memory initialization?

PROBLEM 4 SOLUTION

Once it was clear that memory was not being zeroed, the user (whose iPDS system had the optional dual processors) followed these steps to identify what was preventing memory from being initialized.

```
*ORG 419                        ;Sets address for assembly to 419H
ASM IP= 00419H
*ASM JMP 0A00                   ;Inserts instruction to jump to A00H
ASM IP=00419H       E9E405
*DASM 419                       ;Disassembles instruction at 419H
00419H              JMP     0A00H            ; SHORT E9E405    DASM
*ORG 0A00                       ;Sets address for assembly to A00H
ASM IP=00A00H
*ASM MOV BX,52                  ;Inserts MOV instruction
ASM IP=00A00H                   BB5200
*ASM MOV CX,WORD .MAX           ;Inserts MOV instruction
ASM IP=00A03H           8B0E5000
*ASM JMP 41C                    ;Inserts jump back to 41CH
ASM IP=00A07H           E912FA
*DASM 0A00 TO 0A07              ;Disassembles patch
00A00H              MOV         BX,0052H                BB5200    DASM
00A03H              MOV         CX,WORD 0050H           8B0E5000  DASM
00A07H              JMP         041CH        ; SHORT    E912FA    DASM
*GO FROM 400                    ;Emulates beginning at 400H
..DATA_AND_CODE.REVERSE
00420H              MOV         CX,WORD 0050H           8B0E5000  EX
*
```

280105-6

Figure 6. Sample Patch Commands

1. Used the B processor to locate code: The iPDS file display command (@) was used to scroll through the code listing to locate the line of code where memory zeroing began. The line that completed the zeroing operation was also located.
2. Set up a trace point and a breakpoint: After switching to the A processor, the user set a trace point and a breakpoint that began trace at the first line identified in step 1 and caused emulation to break at the other line identified in step 1. (See the appendix for information on EMV-88 commands.)
3. Initiated emulation using the GO command.
4. Examined the trace buffer: The trace buffer showed tha the expected data (FC) was read from program memory.
5. Connected a logic analyzer to an EMV-88 controller test signal: A logic analyzer was connected to the BRK test signal available on the EMV-88 controller module; the signal is useful in triggering a logic analyzer to capture data on the bus.
6. Re-emulated.
7. Examined the logic analyzer display: The logic analyzer display showed that FF was being received by the processor even though FC was being sent.
8. Connected an oscilloscope to the bus: The oscilloscope showed ringing on the bus. (The ringing was traced to a faulty extender card.) The ringing caused bus signals to be near threshold values. Low signals could be interpreted by the processor as high or low. Thus FC (1111 1100) could be interpreted as FF (1111 1111).

PROBLEM 5: DISPLAY UPDATE SIGNAL IS SLOW

(Task 3: Integrate Hardware and Software)

This user developed a system with a display that must be updated every second. However, in each five minute period, the display was updated one time less than it should be. The user needed to determine whether a counter implemented in software was not generating the correct update signal or whether the output from a separate timer board (that incremented the counter) was too slow.

PROBLEM 5 SOLUTION

This is a problem that became evident to the product development team after software and hardware were

```
*IFF=0              ;Clears the interrupt enable flag
*REGISTER           ;Displays current register settings
-----------------------------------------------------------------
                    *REGISTER DISPLAY*

      RAX=0000H          RAH=00H          RAL=00H
      RBX=0000H          RBH=00H          RBL=00H
      RCX=0000H          RCH=00H          RCL=00H
      RDX=0000H          RDH=00H          RDL=00H

      SP=0FFFH     BP=0000H     SI=0000H     DI=0000H
      CS=FFFFH     DS=0000H     SS=0000H     ES=0000H

      IP=0000H

      RF=F002H     OF=0    DF=0    IFF=0    TF=0
                   SF=0    ZF=0    AF=0     PF=0    CF=0
-----------------------------------------------------------------
 *

 -----
```

280105-7

Figure 7. Sample REGISTER Display that Shows the New IFF Setting

integrated. It was unclear whether it was a software or hardware defect. The team employed the following steps with the EMV-88 emulator to locate the source of the problem.

1. Created a counter macro: A macro was created that counted each time the external board sent a signal to a specific input port of the 8088. (See the appendix for information on EMV-88 commands.) The macro also sent a signal to an ouput port when the counter reached the correct count. The team reasoned that if the problem still existed when they used the macro counter, the counter in the prototype software could be eliminated as the source of the problem.
2. Executed the macro and checked the output signal: The output port signal interval was slightly longer than the desired one second interval. Thus, the problem must be caused by the signal input to the counter.
3. Measured the input signal: The input signal was expected to occur at 0.500 second intervals. However, measurements showed that it occurred at longer intervals. It seemed, then, that the count board was to blame. However, examination of the board's specifications showed that the output of the board was ambiguously specified. In one place it gave the timer output as occurring at 0.500 intervals and in other places the interval was specified with a +0.016 second tolerance. So the cause of the slow display update was neither a hardware defect nor a software defect. Rather, to blame were an ambiguous specification and the failure of the designers to look for and to take into account the tolerance of the timer's interval.

PROBLEM 6: READ-ONLY MEMORY IS WRITTEN TO

(Task 3: Integrate Hardware and Software)

One user encountered a situation in which a read-only area of memory was written to during program execution. The EMV-88 user performed the following steps to isolate the error.

```
*BREAK                ;Displays breakpoint settings
-----------------------------------------------------------------

                      *BREAKPOINT SETTINGS*                 TYPE
-----------------------------------------------------------------
BR0= OFF     BR1= OFF     BR2= OFF     BR3= OFF          :location
BRR= OFF                                                 :range
BRB= OFF                               (GO mode only)    :branch
BV=OFF                                 (STEP mode only)  :value

 MO=EX

-----------------------------------------------------------------

NOTE: BC will clear all breakpoints and set MO=EX

NOTE: MO affects BRR and BR0,1,2,3. Legal MO settings are:
DR-data read    DW-data write    DRW-data read or write    EX-execution
IR-IO read      IW-IO write      IRW-IO read or write
-----------------------------------------------------------------
*

-----
```

280105-8

Figure 8. BREAK Display

PROBLEM 6 SOLUTION

1. Set a range breakpoint: By using the BREAK command (or FUNCTION-2), the current breakpoint settings were displayed. (See Figure 8 for a display of EMV-88 breakpoint settings. Also, see the appendix for information on EMV-88 commands.) The breakpoint mode was set to data write (MO = DW), and the range breakpoint was set to the memory range of interest. As a result of these settings, emulation breaks if a data write occurs within the specified memory range.
2. Emulated.
3. Examined the trace buffer: Examined the previous 16 instructions in the trace buffer (by entering PREVIOUS 16). Defective code was discovered.
4. Patched and tested the code. See Problem 2 for an account of patching procedures.

SUMMARY

Early users of the EMV-88 emulator used the emulator to perform the following functions in the early stages of their products; life cycles.

- Exercised hardware and software in real time.

- For a prototype system with large programs, the EMV-88 emulator was used to supplement main development systems.
- Patched around missing sections of code to allow emulation when some portions of code were unavailable.
- Resolved disputes about whether bugs were in hardware or software.
- Tested prototype hardware and software at a remote site.
- Tested early manufactured systems.
- Helped in troubleshooting early customer system returns.

In addition, early users showed that the resources of the EMV-88 software and hardware can be used to cope with a wide variety of debugging problems. The EMV-88 emulator performed the following tasks:

- Made an initial check of prototype hardware.
- Located code that caused the instruction execution sequence to be wrong.
- Devised a way to debug in a multi-tasking environment.
- Identified the reason that memory was not being zeroed.
- Isolated the cause for a counter counting too slowly.
- Located code that was permitting writing to read-only memory.

Finally, early EMV-88 customers also made use of other iPDS plug-in modules. They used other emulation vehicles to debug other portions of their hardware and software that were designed around other Intel processors; they also used PROM programming modules to load their debugged code into their prototype system PROMs.

New users of the iPDS system and EMV-88 emulator are encouraged to make full use of these systems' capabilities and resources to perfect their products. Only some of the capabilities of the EMV-88 emulator and the iPDS system have been described here. Review the iPDS system and EMV-88 emulator manuals to gain full knowledge of the command sets and options.

APPENDIX: SUMMARY OF EMV-88 COMMANDS AND COMMAND CATEGORIES

APPENDIX: SUMMARY OF EMV-88 COMMANDS AND COMMAND CATEGORIES

The EMV-88 emulator is a full symbolic emulator, and hence all commands and displays can be entered symbolically. The EMV-88 emulator and the user can thus communicate by referring to symbols defined in the user's source program or symbols defined during the debugging session. Ths following sections describe the command categories and Table 1 summarizes the EMV-88 commands.

UTILITY COMMANDS

Utility commands performs functions not directly related to the task of emulation and debugging. These commands gain access to the iPDS system resources and display information about the emulator.

DISPLAY/MODIFY COMMANDS

These commands change or display any register, port, or memory addressable by the iAPX-88 target system. They provide access to specific areas of the processor or target system and thus minimize extraneous display information.

EMULATION COMMANDS

Commands that control program execution or initiate emulation fall into this category. The GO, BREAK, and TRACE commands are in this category.

ADVANCED COMMANDS

The advanced commands offer an easy way to increase the debugging capability of this product. These advanced features allow the EMV-88 emulator command sequences to be combined, executed, and stored.

Table 1. Summary of EMV-88 Commands

Command Category	Command	Command Definition
Utility Commands	DEFINE	Defines symbol or macro.
	DOMAIN	Establishes default module.
	ENABLE/DISABLE	Controls expanded display.
	EVALUATE	Evaluates any expression.
	EXIT	Terminates EMV-88 session.
	HELP	Displays command syntax.
	INCLUDE	Loads a macro definition or a command file.
	LINE	Displays statement numbers and associated absolute addresses.
	LIST	Generates copy of emulation work session.
	LOAD	Loads object file in mapped memory.
	MODULE	Displays module names in EMV-88 module table.
	REMOVE	Deletes symbol or macro.
	RESET	Resets emulation processor.
	SAVE	Saves memory to file.
	SYMBOLS	Displays symbols.
	SUFFIX/BASE	Sets input and displays numeric base.
	TYPE	Sets/displays data type for symbol name.

Table 1. Summary of EMV-88 Commands (Continued)

Command Category	Command	Command Definition
Emulation Commands	BR	Displays breakpoint menu.
	BR0, BR1, BR2, BR3	Breakpoint register for execution address.
	BRB	Breaks on branch.
	BRR	Breakpoint register for execution range.
	BV	Breaks on value.
	BC	Clears all breaks.
	DTR	Displays trace menu.
	GO	Enters real-time emulation mode.
	MO	Break qualifier.
	PREVIOUS	Displays execution trace.
	STEP	Enters slow-down emulation mode.
	TD	Enable/disables display of code disassembly.
	TR	Enable/disables display of registers.
	TR0, TR1, TR2, TR3	Enable/disables display by execution address.
	TS	Enable/disables display of PSW.
	TV	Enable/disables display by register value.
Display/Modify Commands	ASM/DASM	Changes/displays code memory in assembly language mnemonics.
	ORG	Sets address for assembling instructions.
	DUMP	Displays memory as ASCII and hexadecimal.
	MEMORY	Displays menu for memory access.
	PORT	Changes/displays ports.
	REGISTER	Displays 8088 registers menu.
	BYTE WORD POINTER SINTEGER INTEGER	Change/display memory.
	REAL TREAL DREAL	8087 commands
Advanced Commands	DIR	Displays names of all available macros.
	FUNCTION	Invokes macro assigned to function key.
	MACRO	Displays macro text.
	MAP	Sets/displays memory map.
	PUT	Stores macro definitions.
	WRITE	Evaluates and displays expressions and strings.

	IF THEN
	COUNT
	REPEAT
	WHILE
	UNTIL
		Control constructs.